AFRL-RI-RS-TR-2016-030

# FORMED: BRINGING FORMAL METHODS TO THE ENGINEERING DESKTOP

BAE SYSTEMS

*FEBRUARY 2016*

FINAL TECHNICAL REPORT

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

# NOTICE AND SIGNATURE PAGE

| REPORT DOCUMENTATION PAGE | *Form Approved* **OMB No. 0704-0188** |
|---|---|

| | | |
|---|---|---|
| **1. REPORT DATE** *(DD-MM-YYYY)* FEB 2016 | **2. REPORT TYPE** FINAL TECHNICAL REPORT | **3. DATES COVERED** *(From - To)* NOV 2013 – NOV 2015 |

| | |
|---|---|
| **4. TITLE AND SUBTITLE** FORMED: BRINGING FORMAL METHODS TO THE ENGINEERING DESKTOP | **5a. CONTRACT NUMBER** FA8750-14-C-0024 |
| | **5b. GRANT NUMBER** N/A |
| | **5c. PROGRAM ELEMENT NUMBER** 63781D |
| **6. AUTHOR(S)** Howard Reubenstein, Greg Eakman, John Wiegley, Panagiotis Manolios, Mitesh Jain | **5d. PROJECT NUMBER** ASET |
| | **5e. TASK NUMBER** 13 |
| | **5f. WORK UNIT NUMBER** BA |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** BAE Systems, 6 New England Executive Park, Burlington, MA 01803 | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** Air Force Research Laboratory/RITA, 525 Brooks Road, Rome NY 13441-4505 | **10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI |
| | **11. SPONSOR/MONITOR'S REPORT NUMBER** AFRL-RI-RS-TR-2016-030 |

**12. DISTRIBUTION AVAILABILITY STATEMENT**

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2016-0322
Date Cleared: 28 JAN 2016

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

FORMED integrates formal verification into software design and development by precisely defining semantics for a restricted subset of the Unified Modeling Language and transforming application models into both an ACL2s formal specification for analysis and Java code for deployment. Correspondence testing verifies consistent translation and executable behavior between the formal and deployed implementations. Key properties addressed include termination, input-output contract satisfaction and absence of null pointer dereferences.

**15. SUBJECT TERMS**

Formal Methods, Software Verification, Model-Based Software

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** |
|---|---|---|---|---|---|
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | | | **WILLIAM E. McKEEVER JR.** |
| U | U | U | SAR | 45 | **19b. TELEPHONE NUMBER (*Include area code*)** **N/A** |

**Standard Form 298 (Rev. 8-98)**
**Prescribed by ANSI Std. Z39.18**

**Table of Contents**

## List of Tables

## List of Figures

## 1 Summary

Despite potential benefits, formal specification is rarely applied as part of software development. Most programming languages do not contain precise semantics to allow reasoning about application correctness. Formal specifications are either only loosely tied to the code that is deployed, or, when code is synthesized, results in an inefficient implementation.

FORMED seeks a middle ground between formal specification and traditional programming. It first defines a formal semantics for UML. An UML program can then be formally analyzed as well as translated to an efficient, deployable implementation. The formal analysis is done by shallow embedding of the semantics in a specification language that is amenable for analysis. The formal specification of the UML program is also executable. We use correspondence testing to validate that the formal model and the implementation have the same executable behavior and to increase confidence that the properties proven with the formal specification also hold in the implementation.

As shown in Figure 1, FORMED starts with a subset of the Unified Modeling Language (UML), including a model-level action language, and extends it with formal semantics. A code generator then translates the UML program into the ACL2s formal specification language, a subset of Common Lisp, and an efficient implementation. Application developers use ACL2s to define and prove key properties of the application, building up libraries of theorems. Since the formal model of the UML program in ACL2s is executable, we can run concrete tests on it. Correspondence testing applies the same test cases to both the ACL2s model and the implementation code to validate that the both exhibit the same behavior. This increases the trust on both the ACL2s and the implementation of the UML model.

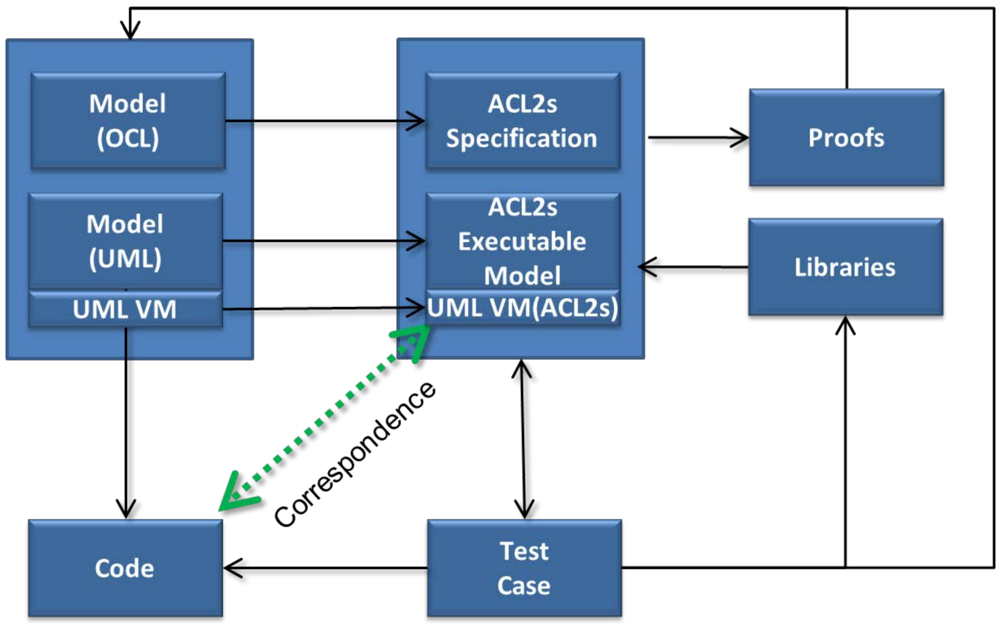

**Figure 1 FORMED generates software specifications and code from a common base model producing corresponding representations of an application**

## 1.1 Task Objectives

FORMED aims to create a software development environment and tools that enable a wider group of software application developers to utilize and exploit the benefits of formal verification. To achieve this, we define the formal semantics of a subset of UML and map it to the ACL2s specification language. We use the ACL2s theorem prover to prove properties of the application.

The tools are packaged within an Eclipse Integrated Development Environment (IDE), including a custom plug-in that allows navigation between the UML and ACL2s views of the application, to aid developer understanding of the mapping.

We apply the FORMED approach to a pedagogical example and a real-world military example to drive development of the toolset and supporting theorems and lemmas.

## 1.2 Technical Problems Addressed

FORMED addresses the technical problems of defining semantics of a subset of UML, transforming UML to a specification language that can be formally be reasoned about, specifying application level theorems and identifying proof strategies to automatically prove these theorems.

In particular, proof automation, represents the biggest challenge. Ideally, the application developer would operate only at the UML level and be completely insulated from the process of proving them. While we have automated some of the theorem proving, formal verification remains a hard problem and, in general, requires the application developer to interact with ACL2s theorem-prover. The FORMED developer often needs to change the contexts between the UML and ACL2s views of the application. FORMED provides an Eclipse plugin that makes this context switching easier.

## 1.3 General Methodology

The FORMED program first provides a shallow mapping from UML to ACL2s. To achieve the mapping, we define a reusable platform model called the MddContext that defines an execution platform, including a stack and heap that support the operational semantics of UML application. We prove a set of theorems that characterize the reasoning about any UML application built on the MddContext.

We then identify properties of the UML semantics that could be used as lemmas in support of application level theorems. We use a pedagogical example of a hotel room lock [1] and a military equipment management protocol to drive the identification of application level theorems to prove.

## 1.4 Important Findings

The shallow mapping from UML to ACL2s provides a rapid way to create a formal specification that conformed closely to the implementation, also derived from the same UML model. The FORMED Eclipse plugin allowed context switching between UML and ACL2s views of the application. A test framework executes test cases against both and compare results, verifying behavioral correspondence and providing a higher level of confidence that the transformations were applied consistently generating ACL2s and implementation code.

The MddContext ACL2s model of the execution platform provides a reusable component, with supporting theorems, for all FORMED UML models, as described in section 4.6.

Using ACL2s, we were able to automatically generate and prove termination of functions, function contracts (including avoidance of NULL pointers), and read-only functions, as described in section 4.15. We automated the generation of other application level theorems, such as uniqueness of object values within a collection and the expression of several other invariants. However, proving these theorems required developers to interact with ACL2s and proof automation for application level theorems remains a challenge.  More application level theorems remained to be proven, and require more platform and UML semantics theorems to support those proofs.

## *1.5 Implications for Further Research*

FORMED provides a baseline for integrating formal verification with software development. With further work in the areas of theorem definitions and developing domain specific proof strategies, the formal verification aspect could be completely hidden from the user while proving a larger set of properties, while still allowing access to ACL2s for advanced theorems.  The support of OCL for UML level theorems and its translation to ACL2s would further hide the formal verification from the application developer.  Completion of the application specific counterexample generation for the heap, described in section 4.9, would also ease the ACL2s learning curve.

We learned that as the implementation of function affects the efficiency of its execution, the definition of function in ACL2s also affects the theorem proving process. Therefore, one should aim to have the simplest possible, and not necessarily the most efficient translation to ACL2s, to improve reasoning and proof automation.  We did not get far enough to determine if these tradeoffs also exist at UML level.

Other theorem patterns, such as reasoning about the implementation’s heap size or the effect of fixed size integers and possible overflow or underflow conditions, could be added to the FORMED theorem templates.  Improvements in the heap and pointer implementations may also improve theorem proving.

The relationship between proofs and testing is also an area for further research.  ACL2s already uses the sub-goals generated by ACL2s in the process of proving a theorem to test the conjecture. We believe these tests can be harnessed to generate a property-directed test-suite and can be used to validate that the implementation also satisfies the same property with a higher degree of confidence. Furthermore, the counterexamples generated by ACL2s from failed proof attempts could be stored for later use as test cases, as they typically represent the corner cases of a function or algorithm.

Research into management of the theorems and lemmas to prove the application level properties would both guide the novice ACL2s user and optimize the verification process.

## 2 Introduction

Formal methods are rarely applied to software development. Current software development methodology focuses on a managed and monitored process and testing to ensure a high quality product. Software development requires tasks that are schedulable and can be measured for progress. And software must run efficiently in time and memory, especially for embedded systems applications.

Formal models, when used, are often disconnected from the fielded implementation. Initial models are created and analyzed to generate requirements, but the connection back to the model is through manual implementation and traceability matrices. Producing an efficient implementation from a formal specification directly is also exceedingly difficult. Specware [2], for example, uses successive refinement to synthesize an implementation from a specification, but the refinement is a manual, multi-step process.

Other approaches require formal models to be complete and sound before moving to implementation can even begin. Building a proof structure is an all-or-nothing process that is difficult to quantify, predict, and schedule. The proof structure can be unstable with respect to changing requirements, or even changing understanding of requirements.

FORMED adds formal methods to the software engineering workflow by adding formal executable semantics to a high-level design language many developers already know, UML. Development remains predictable (as predictable as software projects can be anyway) while allowing formal verification of a specification closely related to the deployed implementation.

Other works in this area have focused on analyzing the semantics of UML itself, such as state machines or activity diagrams. These projects have not addressed how to create software applications that can be formally analyzed.

This report assumes the reader has some background in general software engineering concepts, object oriented design, UML, and formal verification, but need not be an expert in any of those areas.

## 3 Methods, Assumptions, Procedures

### *3.1 Tools*

FORMED combines formal verification with model-based software development. FORMED integrates separate tools under the Eclipse framework, providing developers an IDE that allows them to seamlessly transition between software and verification tools. The tools were selected based on availability, support for key standards, integrability, and usability.

The UML editor, Topcased, is an open source tool that supports both UML and SysML OMG standards, and operates under Eclipse. PathMATE is a commercial UML code generation tool that integrates with multiple UML editors, including Topcased. PathMATE supports an implementation of the UML action semantics meta-model, called PAL, although it does not support the ALF [3] action language syntax that is now also an OMG standard. ALF does not yet have a complete implementation, so PAL was chosen as the model level language. Both ALF and PAL conform to the UML standard for action semantics. PathMATE is also an Eclipse based tool.

We chose ACL2s, the ACL2 Sedan, for several reasons. First of all, it is based on ACL2, which provides the theorem proving power and libraries of ACL2. Second, ACL2 is executable. That helps immensely with validating the formal semantics, e.g. by allowing us to check for correspondence between UML code and the translated ACL2s code by executing both on the same input code. Third, UML is typed and we wanted a powerful mechanism for not only describing types, but also automatically inferring and proving type-like theorems. The ACL2s *defdata* framework allows us to define a rich collection of types and the ACL2s *defunc* macro allows us to define typed functions. Fourth, the ACL2s counterexample generation capability would allow us to find modeling errors automatically. Fifth, ACL2s runs its editor and sessions under Eclipse.

We initially researched Octopus as the tool for translating OCL, but discovered that it was no longer supported. The Eclipse Modeling Framework (EMF), Eclipse's library of support for UML that both Topcased and PathMATE use, includes support for parsing OCL, so we targeted that for integration. However, once we translated UML to ACL2s, we could write our own theorems directly in ACL2s. We chose to focus on theorem proving as a higher risk task, rather than implementing the OCL to ACL2s translator.

Finally, we developed the FORMED plugin to integrate these tools more closely. FORMED generates the UUIDs of model elements into the ACL2s code, and the plugin uses the (Universally Unique Identifiers) UUIDs to map the ACL2s functions and theorems back to the source UML model element. This mapping allows fast transitions between UML and verification views, to both learn the mapping patterns and gather information for proofs.

## *3.2 MddContext*

We needed to adapt the imperative UML action language into the function language of ACL2s. In addition, we needed to model a stack and a heap to capture the state of the application, which called the MddContext. The MddContext is passed around to all ACL2s statements to capture the changes to the state in a functional way.

The first implementation of the context used a single threaded object, *stobj*, to store the application state. However, the *defdata* construct was much more flexible as an implementation.

Section 4.6 describes the context in more detail.

## 3.3 UML to ACL2s Translation

PathMATE provides the capabilities to translate a UML model into various textual forms, including code, documentation, schemas, etc. PathMATE uses a template language to navigate the UML model, extract information and insert that information as text, mixed with hardcoded test of the template. For example, the template code below navigates all the packages in the model, then all the classes in each package, and outputs the class name and description, followed by the class' attributes' names and descriptions. Text enclosed by square brackets "[]" either navigates the model or extracts model information. Text not enclosed is generated directly into the output.

```
[FOREACH package IN system.domains]
Class and Attributes report for [package.name] generated on [date]
[  FOREACH class IN package.objects]
================================================================
[class.name]
description: [class.description]
[    FOREACH attr IN class.allAttributes]
[attr.name]  ([attr.dataType.name])
        description: [attr.description]
[    ENDFOREACH /* attr */]
[  ENDFOREACH /* class */]
[ENDFOREACH /* package */]
```

**Figure 2 PathMATE templates generate code and reports from UML models**

PathMATE comes with templates that generate Java, C, and C++ code, as well as reports. We wrote new templates that generate the ACL2s functions and theorems, but reused some of the provided template that supported model navigation.

## 3.4 Application to Examples

We created example UML models, one of a hotel room's door lock and the hotel systems around it, and a partial model of a military equipment management system. After creating the model, we used an iterative process, cycling between generating the ACL2s, attempting to get ACL2s to accept the functions for the executable parts, trying to prove generate theorems, and fixing the model, transformation rules, or adding new theorems, as needed.

Initial repair work was more on getting the initial transformation templates correct. Then, as we did proofs of our theorems, we discovered additional semantics or lemmas that needed to be defined, either about the MddContext or derived from the UML models. Often we identified one lemma required for one proof using a model element, generalized that lemma in a template, and regenerated the lemma for every model element of that type. Those new lemmas were useful in other proofs.

This generated more lemmas than perhaps were needed, since not all model elements were involved in a proof. No attempt was made to find the optimal set of theorems and lemmas.

## *3.5 FORMED Pair Programming Process*

We envisioned FORMED as a way to rapidly create a formal specification that closely conformed to an implementation, using patterns expressed as templates. We assumed the initial application developer would have little ACL2s experience, but would be paired with a formal verification engineer/ACL2s expert. The verification engineer should not have to become an expert in the subject matter being developed. Initially, Northeastern served the verification engineer role while BAE Systems engineers played the part of novice ACL2s users.

During the second half of the project, BAE added a verification engineer familiar with other languages and tools, such as Coq. While the language was easy to pick up, based on Lisp, the tools took a while to learn. Documentation seems to be written for other researchers, rather than for novices. Reading through the documentation a couple of times while ascending the learning curve helps the novice to pick up knowledge that requires hands-on experience to put together.

## 4 Results and Discussion

This section describes in more detail the work performed, tools used, tradeoffs made, and results collected.

## *4.1 UML as a Development Language*

FORMED uses UML as a high-level language for software development, using both graphical and textual representations to capture software structure and behavior. The FORMED UML profile builds on work to strengthen UML's semantics to make it an executable specification language [4]. Through code generation, FORMED produces deployment code. FORMED uses the PathMATE toolset [5], a commercial product, to provide deployable code generation.

While not specific to the UML profile, one of the key benefits of the FORMED approach is the separation of application concerns into separate UML packages. Each package models the problem space of the area, focusing only on the rules, policies, and behaviors of that area. A façade pattern wraps the package in an API minimizing coupling between packages. This also enables reasoning about the correctness of packages to be verified separately, and composed to build the application, as described in section 4.13.

As verification engineers are scarce resources, FORMED envisions pairing application developers with verification engineers as a team to do software modeling and formal verification. Rather than requiring the verification engineer to become an expert in the application domain, the developer creates the UML model and translates it to ACL2s. The verification engineer then uses the ACL2s theorem prover to prove correctness and safety properties.

Other studies map a subset of a development language to a formal language. Lambda S5 verifies Javascript code through a similar shallow mapping transformation [6]. They describe their process as semantic altering transformations, a bit tongue-in-cheek, but they describe restrictions on the use of features within the source language. For example, Javascript functions can also be treated as objects, but at a cost in Lambda S5.

## *4.2 ACL2s*

We chose ACL2s, the ACL2 Sedan, for several reasons. First of all, it is based on ACL2, which provides an applicative programming language to model systems, a logic and a powerful theorem prover to reason about them. Second, ground expressions in ACL2 are executable. That helps immensely with validating the formal specification e.g. by allowing us to check for correspondence between UML code and the translated ACL2s code by executing both on the same inputs. Third, UML is typed and we wanted a powerful mechanism for not only describing types, but also automatically inferring and proving type-like theorems. The ACL2s *defdata* framework allows us to define a rich collection of types and the ACL2s *defunc* macro allows us to define typed functions. Fourth, we wanted the ability to find bugs in our models. After all, during the design process, models almost always contain errors that we want to quickly find and correct. The ACL2s counterexample generation capability allowed us to find errors automatically.

ACL2s is taught to freshmen in Northeastern's computer science program [7], so it is not unreasonable to expect software developers to be able to pick it up and it is based on Lisp, an executable programming language.

ACL2s also provides automated proof of termination when admitting functions. Proving termination is a precondition to other proof properties. Only functions derived from *while* loop action language could not have termination proven, because the loop variables were hidden on the MddContext stack rather than ACL2s variables or parameters.

### 4.2.1 Encapsulation

Defdata records are similar to structs in C. Defdata provides no support for encapsulation or information hiding. For example, MddContext includes an attribute heap representing heap memory. Access to the heap goes through the API functions. A set of theorems proves that the other MddContext API functions do not change the heap, but only addToHeap, rmFromHeap, and writeAttr change the heap. However, since the MddContext, without information hiding protections, is passed around, separate lemmas must be written to prove that other functions do not alter the heap.

## *4.3 FORMED Eclipse Integration*

FORMED provides an integrated workbench for high assurance software, integrating software design and formal verification tools under the Eclipse integrated development environment (IDE).

Software developers model the system using the Unified Modeling Language standard from the Object Management Group, and generate application code and ACL2s specification code at the same time. With help from a verification engineer, theorems about the application model are developed and proven using the ACL2 Sedan. A FORMED plug-in provides an Eclipse perspective (an arrangement of windows within the IDE) and the ability to navigate between the ACL2s code and the UML model. A JUnit project provides the framework for correspondence testing to show that both executable versions of the model, Java and ACL2s, behave equivalently for all defined test cases.

## *4.4 Examples*

We use 2 examples to check the development of FORMED and to drive the identification of theorems, lemmas and proofs. The first example models a hotel, the management of programmable card keys, and the protocol for keeping keys in sync with the room locks while guaranteeing security for guests.

The second example implements a portion of a military protocol for remotely managing equipment in the field. This example uses UML state machines to manage unreliable communications between the equipment and the management server.

## *4.5 UML to ACL2s Mapping*

The Unified Modeling Language (UML) is a standard object-oriented software design notation created and supported by tool vendors [8]. The semantics of UML were left weak by design, to allow tools to support the array of UML dialects that could not be agreed upon in the standard. However, the standard allows profiles to extend the language with custom semantics and create domain-specific languages based on the notation.

The FORMED UML profile uses a subset of UML model elements and restricts the usage of these elements, while enhancing the profile with executable semantics. The FORMED profile consists of UML classes, properties, operations, and associations. The profile also includes flat finite state machine constructs to capture lifecycle and asynchronous behavior. A model level action language, conforming to the UML meta-model for action language [3] and with syntax similar to Java, captures the detailed behavior of the model. FORMED uses the PathMATE UML code generator to map the UML model elements to ACL2s.

FORMED performs a shallow mapping of UML into ACL2s. This mapping is automated with a customizable transformation tool that is configured to generate ACL2s, and make use of the defdata and defunc constructs described in the next sections. The FORMED transformation generates ACL2s executable code based on the UML executable semantics and also generated auxiliary functions and lemmas for each model element's semantics and application level theorems, such as invariants.

The generated ACL2s constructs run atop the MddContext, a platform model for managing the applications state, described in MddContext Platform Model section 4.6. The tables below summarize the mappings of UML model elements to ACL2s constructs for class diagram elements (Table 1), state machines (Table 2), and UML action language (Table 3). Implemented constructs are shown in green, partially implemented in yellow, and not implemented in red.

Table 1 UML class diagram elements mapped to ACL2s constructs

| Class Diagram | | | |
|---|---|---|---|
| **Model Element** | **ACL2s Implementation** | **Status** | **Comment** |
| Façade Component | ACL2s module | Done | |
| Attribute | record entry within defdata definition | Done | |
| Class | Defdata - data definition, with records. | Done | |
| Data Types | Base ACL2 data types, integers, booleans, defdata for enumerations | Done | Supported in translation, proofs based on rational numbers not performed. |
| Association | defdata record.  Plus link, unlink, and navigation functions. | Done | |
| Associative Class | | Not Done | Can be worked around using multiple associations. |
| Generalization/ Specialization | Type record in every generated class | Done | |
| Class Operation | ACL2s function | Done | |
| Façade Operation | Façade function | Done | |
| Parameter | function parameter | Done | |
| Composition/ Aggregation | None | Not Planned | Association decorations may have some semantic meaning that can be used in proofs. |

Table 2 UML state diagram elements mapped to ACL2s constructs

| State Diagrams | | | |
|---|---|---|---|
| **Model Element** | **ACL2s Implementation** | **Status** | **Comment** |
| State | Enumeration of states, state attribute in defdata record. | Done | |
| Transition | Either as a lookup table or complex condition statements | Done | |
| Entry/Exit/Transition Actions | | Done | |
| Ignored Events | | Done | |
| Guards | | Not Planned | |
| Deferred Events | | Not Planned | |
| Nested States | | Not Planned | |
| History | | Not Planned | |

Table 3 UML action language statements and expressions mapped to ACL2s constructs

| **Action Language** | | | |
|---|---|---|---|
| **Model Element** | **ACL2s Implementation** | **Status** | **Comment** |
| Create | constructor provided by defdata | Done | |
| Delete | Destructor function that cleans up links across associations. | Done | |
| Find | Separate recursive function | Done | |
| Foreach | Separate recursive function with loop block as implementation and local context passed in. | Done | |
| Link | Call to association function to add to list | Done | |
| Unlink | Call to association accessor to remove from list | Done | |
| Navigation | Call to association accessor access list | Done | |
| Downcast Navigation | Base function that checks object's classes against requested downcast class and returns nil if not matched. | Done | |
| Attribute Access - read | mget record access plus update to context. | Done | |
| Attribute Access - write | mset record access plus update to context. | Done | |
| While loop | Separate recursive function with loop block as implementation and local context passed in. | Partial | Translation supported but not proof automation. Challenge: detection and specification of termination conditions. |
| Service Handle | | Done | |
| Sort Objects | | Not Planned | |
| If | | Done | |
| Return | | Partial | Only supported at end of operation. Reviewing continuation passing style (CPS) pattern. Current requirement is single-entry-single exit operations. |
| Break | | Not Planned | |

| | | | |
|---|---|---|---|
| Local Variable | | Done | |
| Binary Operators | | Partial | Math, equality operators complete, missing bit shift operators (see ACL2 ASH) |
| Unary Operators | | Done | |
| Cancel Event | | Done | |
| Generate Event | | Done | |
| Operation Call | direct function call, creating new stack context | Done | |

## 4.6 MddContext Platform Model

FORMED creates an abstract ACL2s model of the application execution environment, called the MddContext. The context models the computing platform, independent of target applications, and is reused across all FORMED analysis. This section described the MddContext, its semantics, its API, and some of its theorems.

The FORMED profile's object-oriented (OO) semantics presented a challenge in mapping to a functional language like ACL2s. Functional languages define functions purely by their inputs and outputs, and there is no internal state maintained. OO semantics allow for a global state represented by a heap, but global state is not permitted in functional languages. Similarly, we convert iterations to recursive functions, but the recursive functions need to operate on the same stack as the function that it is called from. This led to the modeling of the application's global state in a defdata construct called MddContext, shown in the code below. This context is passed through every statement, using a *seq* macro to make the generated code look more like sequential programming statements.

```
(defdata HeapMemory (map address all))
 (defdata MddContext-type
  (record (heap . HeapMemory)
         (nextAddr . nat)
         (csstack . stack)
         (pendingQueue . PendingEventQueue)
         (curTime . nat)
         (cmdLineArgs . stringList) ))
```

**Figure 3 MddContext contains the application's state information and is updated by each line of action language**

The context consists of a heap, stack, a next address counter, a clock, an event queue, and list of command line arguments. The heap represents application memory, but in a simpler way, as a mapping of an address (a natural number) to an object in memory. The MddContext is a reusable platform, so the types of objects on the heap are deferred, as shown in the definition of HeapMemory above, mapping the address to the *all* data type. FORMED refines the HeapMemory to <application>HeapMemory , a list of UML-derived objects for each application, as shown below, with a theorem showing it as a subtype of the HeapMemory and convenience function to be used in other theorems.

```
(defdata <application>HeapMemory (map address SystemHeapObjects))
(defthm heap-subtype
  (implies (HotelDCHeapMemoryp heap) (HeapMemoryp heap))
  :rule-classes (:tau-system))
(defunc constrainedHeap (MddContext)
  (HotelDCHeapMemoryp (mget :heap MddContext)))
```

**Figure 4 FORMED refines the heap definition per application**

Access to the data on the heap first goes through the address to access the object that maps to it. Then, attributes within the object may be accessed. There is no direct memory access to internal attributes like C++ or Java memory models.

The stack component of the MddContext models the stack of a C++ or Java execution platform, providing a "scratch pad" of local variables for operations to perform their computations. UML operations map directly to ACL2s functions, which could use ACL2s *let* and *let** constructs for defining local variables. However, ACL2s, as a functional language, does not support iteration, but required iteration to be expressed as recursion. Thus, UML action language constructs such as FOREACH and WHILE must be mapped to recursive functions whose implementation is the loop body. Those recursive functions would not have access to the ACL2s stack of the function containing the loop. The MddContext stack allows functions to share the local variables between the operation functions and the recursive functions.

The stack maps a string, representing the local variable name, to a value of any type. The stack interface provides access to write and read local variable values. A read returns the last written value, or *nil* if no value had been written.

Each UML operation, as defined by the translation rules into ACL2s, pushes a new stack entry onto the stack for its own local variables at the beginning of the function and pop the stack at the end of a function. Operations only have access to the top of the stack, and cannot access the stack of other functions. There is no ACL2s language support for encapsulation, so hand-written ACL2s code could be inserted to alter stack entries of other functions. Theorems prove that this does not happen when using the MddContext API.

The required push/pop semantics of the stack, as well as functional language constraints, are the reason the UML action language constrains the use of the RETURN statement to the end of the function. FORMED UML operations are single entry, single exit.

The eventQueue supports finite state machine execution (see section 4.14) and contains the set of events that have been sent but have not yet been processed. The eventQueue delivers events to objects, which are received and processed according to the object's state machine. Events resulting from actions triggered by handling the event are placed on the eventQueue for delivery and processing. (See section 4.14 for more on state machine semantics)

The curTime attribute of the MddContext represents a virtual clock that also relates to finite state machine execution, but only to support proper event ordering. State actions and functions in FORMED are all run-to-completion and assumed to take zero time. The FORMED profile, based on the underlying PathMATE tool, allow events to be sent with a delay, meaning that when the event is processed, the MddContext time must be updated to enable other delayed events in the correct order.

The MddContext attribute cmdLineArgs represents the list of strings passed to the application via the command line. ACL2s does not have access to command line arguments the way C++ and Java do, so cmdLineArgs provides the equivalent behavior. This is supported mainly for conformance testing. Access to the arguments is provided by the GetCommandLineArgs operation of the SoftwareMechanisms package. This function is not generated, but hand implemented, and is described in section 4.13.

The table below defines the API functions that operate on the MddContext, grouped by the part of the context they operate on. A separate (large) set of theorems describes the properties and interrelationships of these functions.

**Table 4 MddContext API functions**

| Function | Parameters | Return Value | Description |
|---|---|---|---|
| Stack Functions | | | |
| ctxpush | fnName MddContext | MddContext | pushes stack for a new operation |
| ctxpop | MddContext | MddContext | pops stack when leaving operation |
| ctxHead | MddContext | Entry | returns the current stack |
| ctxsetvar | varName val MddContext | MddContext | adds name-value pair to stack's alist |
| ctxgetvar | varName MddContext | Val | returns top alist entry for varName |
| Heap Functions | | | |
| addToHeapLocalVarAddr | val addrName MddContext | MddContext | adds object to heap, address on stack |
| getFromHeap | addr MddContext | Object | get an object from the heap |
| rmFromHeap | addr MddContext | MddContext | delete an object from the heap |
| classExtent | type MddContext | objList | Get UML class extent from the heap |
| readAttr | attrName addr MddContext | Val | Gets heap object's defdata record |
| writeAttr | attrName addr val MddContext | MddContext | Updates the heap object's defdata record |
| Association Functions (also accesses heap) | | | |
| addToManyAssoc | container assocName obj | MddContext | inserts an address into the association list of an object |

| | | | |
|---|---|---|---|
| | MddContext | | |
| rmFromManyAssoc | container assocName obj MddContext | MddContext | removes an address from the association list of an object |
| setSingleAssoc | container assocName obj MddContext | MddContext | sets the address of an association pointer of an object |
| clearSingleAssoc | objList assocName MddContext | MddContext | clears the address of an association pointer of an object |
| rmFromOtherSides | rmObj objList assocName MddContext | MddContext | removes an address from other objects that point to it |
| unlinkFromAll | srcObj assocName otherSides isSingle MddContext | MddContext | removes an address from other objects that point to it |
| traverseNav | src assocName MddContext | resultList | returns a list of pointers related to the src objects across accosName |
| Finite State Machine Functions | | | |
| enqueueEvent | ev selfDirected dest src fireTime params MddContext | MddContext | Creates and adds the event to the MddContext pendingQueue |
| getPossibleNextEvents | MddContext | PendingEvenetQueue | Returns a list of the context's pending events that can be handled next |
| getNextEvent | index possibleList | matbeEvent | Returns the event in the possibles list at the index, or nil if the index is out of range of the list is empty |
| removeEvent | ev MddContext | MddContext | Removes the event from the pendingQueue, with the expectation that it will be handled |
| handleEvent | pendingEvent MddContext | MddContext | Generated function that delivers the event to the class function that handles it |
| cancelEvent | ev dest MddContext | MddContext | Deletes the next delayed event to the destination |

The context provides a reusable platform for executing the UML application models in ACL2s. This platform is reused across all UML models, but may also be generic enough to support other DSL mappings.

The MddContext is similar to the Stateman state manager platform developed to represent hardware byte addressable memory and program counters [9].

## *4.7 Extents and Pointers*

The FORMED UML profile includes the concept of class extents, where a class keeps track of all its instances. While we could have set up additional bookkeeping to track these, instead we chose to query the heap every time an extent is requested. The *_umlTypes* attribute, added to each defdata derived from a UML class, tags the heap object with its type, and since the heap only contains objects of the UML classes, we can easily query the heap for a class' extent, returning an *objList* where every member points to an object in heap of the specified type.

Additionally, for each UML class we generate a helper function that verifies that given a list of pointers to objects in the heap, the objects referenced by each pointer is an instance of the particular UML class. These verify functions are called whenever a class extent is referenced or an association traversed in the action language. These functions are useful in specifying properties of the UML application as well as in runtime validation.

While dynamically rebuilding the extent by querying the heap was a straightforward, maintainable implementation, the approach presented problems with proofs. First, the heap interface, built on top of the ACL2s map data type, consisted of *mset* and *mget* functions whose properties were well defined, but whose implementations were hidden. The opacity of these functions made it difficult to reason about the contents of the heap, requiring us to reason about analogs of the heap. For example, an extent lemma that stated if an object, with a type, exists on the heap, then that object will be returned in the type’s extent. That lemma turned out to be difficult to prove on its own (see section 4.8.3 for the solution).

Alternative implementations, such as implementing the heap as an association list may allow easier proofs, given its transparent implementation as compared to the map. Also the extra bookkeeping involved with explicitly managing the extents as sets of object references in the ACL2s implementation, would have provided additional opportunities for lemmas that dynamically rebuilding the extent did not.

FORMED used a natural number, representing the address of an object, as the key to looking up the object on the heap. However, this did not provide the type safety required by some theorems. There was no guarantee that address 105 would reference a Room, as required by a theorem, and the base functions only guaranteed an object or nil. We generated additional functions and theorems to enforce type safety, checking the type requested against the type of objects retrieved. These functions provided type safety both on extents and association traversals (e.g., from Hotel to all of its Rooms).

We considered, early on, making the address more complex, by adding the type of the object referenced [10]. At the time, we chose simplicity of implementation. We did not have time to experiment with the typed addresses to determine how it impacted type safety and the theorems that required it. (Future work)

Object-oriented inheritance was another concept that needs to be adapted in mapping to ACL2s. The *_umlTypes* attribute was implemented as a list, to account for inheritance trees. Dynamic binding could be simplified during transformation to account for all subtypes of a class – no new subtypes would be added. Thus, subtype operations could override supertype operations, and in ACL2s, the implementation pattern would be to provide an implementation at the supertype that would examine the type of object the operation was called on, and invoke the correct subtype operation.

One area for further work, associated with defdata would be to simulate higher order logic functions, e.g., projection of a set of Person objects to a set of names. This capability would have proved useful for the uniqueness theorems. (Future work)

## 4.8 Theorem Management

This section describes our experiences performing proofs in ACL2s under FORMED. ACL2s is a semi-automated theorem prover that uses heuristics to prove a theorem using other theorems it "knows", and by applying techniques such as rewrite, induction and equivalence. However, ACL2s sometimes requires guidance, in the form of restricting the theorems it applies. Some theorems may be applied incorrectly during the heuristics, leading either to a failed proof or non-termination.

### 4.8.1 Proof versus Programming

Since one of the primary objectives of the FORMED project was to make proof engineering accessible to regular programmers, it was natural for us to approach proof construction from a programming mindset. For example, in many cases our choice of data representation or algorithm was typical of what a software engineer might choose. We intentionally proceeded along lines that would be familiar to the average programmer.

However, what we discovered is that although proof systems can be made more accessible, they do require a shift in mindset to accommodate the differing needs of proof. As an example: one of our fundamental data structures was an object heap. We chose a mapping structure from indices to objects, and allowed objects of any type to be stored in the heap.

This worked well for quite some time. However, as we started trying to prove more complex theorems concerning alterations of the heap (see section 4.7), this choice of data structure, and its inherent dynamism, began to hinder our work significantly.
The mapping structure, while more efficient than a typical association list, did not allow reasoning about its internal structure. This meant that properties such as the ordering of elements, or the set of all possible elements in the heap, could not be used conveniently in proofs.

Likewise, our choice of a heterogeneous heap led to many complications. For example, showing that an object of a certain type in the heap must be returned in a search for all objects of that type proved impossible to demonstrate using just the heap and its interface functions, mget and mset. Had we separated the heaps by object type, this sort of proof would have been trivial.  Instead, we had to use a technique defining equivalent functions, described in section 4.8.3 below.

These experiences have led us to realize that **crafting a model suitable to proof is a different thing from crafting an efficient program**. Sometimes, the worst possible choice for an ordinary program, can lead to the simplest and most efficient proofs. As long as there is a proven equivalence between these two domains, there is no harm done, except that the FORMED user must be aware of the differing needs of each domain.

### 4.8.2 Performance

Since the primary goal of proof engineering is to produce evidence of a property, it can be easy to overlook orthogonal concerns like proof performance.  However, on the FORMED project we discovered that a little attention paid to such matters can go a long way.

ACL2s includes instrumentation to help determine which theorems are useful during proofs, and which lead to “dead ends”.  The ACL2s construct *accumulated-persistence* enables this instrumentation, and counts how many times each theorem is applied and how many times it has led to successfully solving a goal or subgoal.

For example, a proof of one particular theorem took ten minutes to pass in the end. Considered on its own this is acceptable, but those 10 minutes were repeatedly paid as we tested each new helper lemma. Had we used the "accumulated-persistence" mechanism from the beginning, to determine why it was so slow to process, it would have greatly sped up the development and test cycle. Later applications of the technique proved this to be true.

Another method that was sometimes employed was to prove theorems in the least environment possible, disabling most theorems, to avoid any performance impact from ACL2 knowing too much. The greater the number of theorems enabled, the more ACL2s has to go through to find a working proof structure.  After the theorem had been proven, it was copied into the destination context and reproved there. It still took much longer to execute in the final location, but no cumulative cost was paid to develop it.

Theorems may include hints to instruct ACL2s on which theorems should be enabled or disabled for the proof.  While this can optimize a stable proof, it does not provide flexibility when updating the model, under maintenance, for example.  Attempts to group the generated theorems together to easily enable and disable them did not ease the effort.

A secondary consequence of paying attention to performance from the outset is that if most theorems pass quickly, any attempted theorems that take too long will likely never pass -- or time should be spent on performance analysis before continuing.  Even if a proof passes when run overnight, this communicates a need to improve the environment, more than the mere result of the proof itself.

### 4.8.3 Equivalent Functions

The FORMED MddContext heap was built on the ACL2s map construct, with simple, but opaque, API functions mset and mget, to add and retrieve objects using integer keys. The implementation of mset and mget functions is very efficient but difficult to reason about. ACL2 provides a set of rewrite rule that are often sufficient for reasoning about functions defined using mset/mget procedures. While the map provided the basic theorems for its operation, it did limit our ability to reason about its contents.

In one example theorem, if an object of type A exists on the heap, then it should be returned in the extent of A, computed by searching the heap for objects of type A. To prove this, we defined *get-value*, a function that provides functionality on MddContext heap similar to what mget provides for a map. We:

- proved the get-value function behaves equivalently to mget,
- proved that the object would be reported in the extent of type A if using get-value to search, then
- proved that the object would be reported in the extent using mget

Section 4.7 describes more on extents and pointers, and the suggestion that we swap out the heap's map implementation with a less efficient, but more proof-friendly alist.

### 4.8.4 History variables

Often state of a system is augmented with history variables that record a sequence of past values that some state variable took in the past. History variables do not change the behavior of the system in any way, but greatly reduce the proof effort.

For example, proving a uniqueness theorem that all objects in the Person class extent have a unique name could use a list variable *names* to capture the names of the Persons. The list variable would be updated as Persons were added, deleted, or changed their names. The proof would then flow

- the *names* list was a projects of the Person names, using an invariant,
- the application enforces that names in the list are unique,
- therefore the names in the Person extent are unique.

Added history variables in the UML make the implementation less efficient, adding overhead in terms of execution time and memory, although they do not affect the behavior. We optimize out the history variable by marking it as Deleted in the properties.txt file, where the transformation will remove all references to the variable so it does not appear in the implementation. Correspondence testing would show that the efficient Java implementation and less efficient ACL2s would behave the same.

Note that we did not use history variables in our current set of proofs, but defined the approach to using them.

## *4.9 Counterexamples*

Counterexamples provide important feedback, especially to novice ACL2s users, on defects in conjectures or functions. The example can be used to reverse engineer a failing test case that can be used to understand and fix the underlying problem.

Indeed, for developing and proving theorems about functions and not operating on the MddContext application state, counterexamples were very useful.

The initial hand-coded ACL2s model of the hotel lock and key (Figure 5) consisted of just the core logic of determining whether to unlock the door and whether to update the lock state. The core safety theorem, that the previous key could not unlock the door once a new key was used successfully (Figure 6).

```
(defun unlock (lock cardCodes)
  (let ((cardCode1 (mget :curCode cardCodes))
        (cardCode2 (mget :prevCode cardCodes))
        (lockCode  (mget :code lock)))
    (cond ((equal lockCode cardCode1) t)
          ((equal lockCode cardCode2) t)
          (nil)) ))

(defun updateLockCode (lock cardCodes)
  (let ((cardCode1 (mget :curCode cardCodes))
        (cardCode2 (mget :prevCode cardCodes))
        (lockCode (mget :code lock)))
    (cond ((equal lockCode cardCode1) lock)
          ((equal lockCode cardCode2) (RoomLock cardCode1))
          (t lock))))

;; returns the new lock state and if the door should unlock
(defun enterRoom (lock cardKey)
  (cons (updateLockCode lock cardKey) (unlock lock cardKey)))
```

**Figure 5 Hand-written model of hotel lock logic to update the lock state and decide to unlock.**

```
(defthmd next-key-unlocks-prev-key-denied
  (let* (; guest1 enters with key1
         (g1CanEnter (Hotel_Lock_checkKey lock key1 MddContext))
         ; guest2 enters with key2, which succeeds key1
         (g2CanEnter (Hotel_Lock_checkKey lock key2 (mget :MddContext g1CanEnter)))
         ; guest1 tries to enter
         (g1CannotEnter (Hotel_Lock_checkKey lock key1 (mget :MddContext g2CanEnter))))
        (implies (and (MddContextp MddContext)
                      (natp lock)
                      (natp key1)
                      (natp key2)
                      (Hotel_Lockp (getFromHeap lock MddContext))
```

```
    (Hotel_CardKeyp (getFromHeap key1 MddContext))
    (Hotel_CardKeyp (getFromHeap key2 MddContext))
    (equal (mget :_retval g1CanEnter) t)
    (equal (mget :_retval g2CanEnter) t)
    ; key2 succeeds key1 (key2.prevCode == key1.curCode)
    (mget :_retval (Hotel_CardKey_succeeds key2 key1 MddContext))
    (not (equal (mget :curCode (getFromHeap key1 MddContext))
            (mget :curCode (getFromHeap key2 MddContext))))
    (not (equal (mget :curCode (getFromHeap key2 MddContext))
            (mget :prevCode (getFromHeap key1 MddContext))))
  (equal (mget :_retval g1CannotEnter) nil))))
```

**Figure 6 ACL2s Counterexamples identified cases (in bold) where the theorem failed – when keys are duplicate or inverse.**

**Counterexample generation (cgen) in ACL2s correctly identified counterexamples when the cards are duplicates of each other (an easy to overlook corner case),** and when the cards are inverse (previous.curCode = next.prevCode and prev.prevCode = next.curCode) causing the lock state to toggle but always permitting either card.

Counterexample generation also helped the development of the MddContext and its reasoning framework theorems.

Cgen failed to produce any counterexamples at the UML application level. Cgen assumes that the most critical values in a nested defdata structure like MddContext appear at the top, whereas the interesting application data exists in the heap, 3 levels deep in the MddContext.

Another counterexample generation problem arose from the data type of the heap, a map from address to *all*, allowing a much wider variation of heap objects than actually allowed by the application. Application specific restrictions on the heap narrowed the types of objects to only those derived from the UML models (described in section 4.6).

Finally, cgen could not generate examples that were well-formed with respect to the FORMED UML profile semantics. In particular, the bidirectional referential integrity of associations requires that each object of a pair of linked objects contain a reference to the other. In Figure 7, semantics of association A2 require that the Room have a reference to the Hotel, and the Hotel contains a reference to the Room. Cgen could not generate examples that would meet the bidirectional constraint.

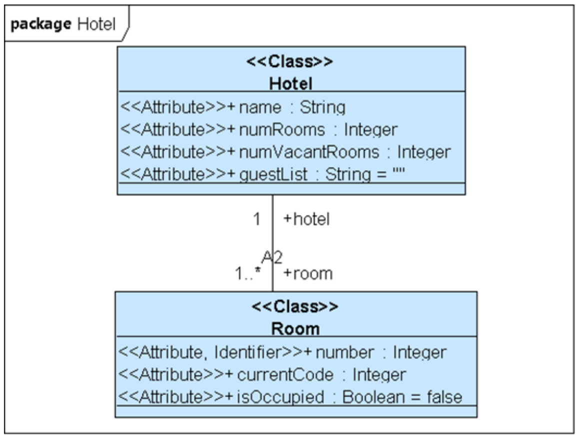
package Hotel
<<Class>>
Hotel
<<Attribute>>+ name : String
<<Attribute>>+ numRooms : Integer
<<Attribute>>+ numVacantRooms : Integer
<<Attribute>>+ guestList : String = ""
1
+hotel
A2
1..*
+room
<<Class>>
Room
<<Attribute, Identifier>>+ number : Integer
<<Attribute>>+ currentCode : Integer
<<Attribute>>+ isOccupied : Boolean = false

**Figure 7 FORMED semantics require bidirectional referential integrity**

Since counterexamples were initially assumed to be an important asset to novice ACL2s users, we designed our own counterexample generation algorithm for FORMED based on the UML class diagram. We integrated with the existing cgen algorithms through the use of a new ACL2s construct, *defdata-attach :enumerator* that allowed us to insert our custom example enumeration algorithm for the heap.

Each UML class generates its own enumeration function that makes use of the ACL2s functions for most of its attributes, but ensures the integrity of the associations. The enumeration function take a natural number as a randomization seed, an MddContext containing the heap being constructed, and an association name, to prevent recursion over an association. The randomization seed is decreased across each association and guarantees termination of the algorithm.

The class enumerator function first uses the enumerator generated by the defdata construct to create the initial object. Then the function resets the association attributes to ensure the bidirectional property is preserved. It then adds the new object to the heap and gets back its address. From there, it creates and links objects across associations, using other class enumerator functions, before returning an MddContext and its modified heap along with the address of the object just created (in case it will be linked by a calling class enumerator function). Two example mutually recursive functions are shown below in Figure 8, but only formalizing one association, and using function fdecr to decrement n.

```
(defun f-nth-hotel_room (n MddContext notAcross)
  (let* ((obj (nth-hotel_room n))
        (obj (mset :acrossA1_to_lock nil obj))
        (obj (mset :acrossA2_to_hotel nil obj))
        (obj (mset :acrossA7_to_currentOccupant nil obj))
        (addr-heap-cons (addToHeap obj MddContext))
        (ptr (car addr-heap-cons))
        (MddContext (cdr addr-heap-cons))
        (MddContext
         (if (equal notAcross 'A2)
            MddContext
            (let* ((h-ctx-cons (f-nth-hotel_hotel (fdecr n) MddContext 'A2))
                  (hptr (car h-ctx-cons))
                  (MddContext (cdr h-ctx-cons))
                  (MddContext (link_Hotel_A2 hptr ptr MddContext)))
               MddContext))))
         (cons ptr MddContext)))

(defun f-nth-hotel_hotel (n MddContext notAcross)
  (let* ((obj (nth-hotel_hotel n))
        (obj (mset :ACROSSA13_TO_SIGN nil obj))
        (obj (mset :ACROSSA14_TO_PAYMENTSYSTEM nil obj))
        (obj (mset :ACROSSA3_TO_CARDKEYENCODER nil obj))
        (obj (mset :ACROSSA2_TO_ROOM nil obj))
        (addr-heap-cons (addToHeap obj MddContext))
        (hptr (car addr-heap-cons))
        (MddContext (cdr addr-heap-cons))
        (MddContext
          (if (equal notAcross 'A2)
            MddContext
            (let* ((room-ctx-cons (f-nth-hotel_room (fdecr n) MddContext 'A2))
                   (roomptr (car room-ctx-cons))
                   (MddContext (cdr room-ctx-cons))
                   (MddContext (link_Hotel_A2 hptr roomptr MddContext)))
               MddContext))))
     (cons hptr MddContext)))
```

**Figure 8 Mutually recursive class enumerators create objects to populate heap**

The heap enumerator uses the class enumerators, choosing, based on the seed, one or more starting point classes for the example heap object population.

```
(defun f-nth-HeapMemory (n)
  (let* ((MddContext (initCtx))
         (i (mod n 2))
         (MddContext
           (case i
                 (1 (cdr (f-nth-hotel_hotel n MddContext nil)))
                 (2 (cdr (f-nth-hotel_room n MddContext nil)))
                 (otherwise MddContext) ))  )
        (mget :heap MddContext)  ))
```

**Figure 9 Custom heap enumeration function generates well-formed FORMED object populations to try as counterexamples**

(Future Work) We did not have time to implement this algorithm, so experimentation and evaluation of its effectiveness remains as potential future work.

(Future work) Counterexamples become a transient part of the development process. Developers use them to fix their theorems or functions and move on. Those counterexamples usually expose important corner cases could be captured and reused as test cases.

## *4.10 Process*

The simplified FORMED development process goes from UML to proofs as described in the steps below.

- Model - model-based development captures the software specification in a form that can be used to generate additional software production artifacts
- Test – simple (unit) testing is an understandable and efficient way to make sure the model is in the right ballpark. (Specific test cases are a very simple way to state requirements – albeit incomplete.)
- Proof Plan – Define invariants, pre-conditions, and post-conditions. ACL2s evaluates proof plan and provides counter-examples.
- Prove – only prove properties once initial testing and analysis indicates proof is likely to succeed.
- Test – confirm correspondence of model and code and demonstrate proofs with additional testing (to provide traditional visible evidence)

### 4.10.1 Maintenance

One thing common to all software development is change. Change comes from changing customer requirements, evolving understanding of requirements, new features, and performance tuning, among other sources. Proof structures, however, are remarkably intolerant of change, as shown in the experience paper applying Coq to software development [11].

FORMED incorporates changes into the process by allowing modifications to the source UML model, allowing proofs to fail, so generated code can be tested while rebuilding the proofs in parallel.

Though maintaining proofs while software is undergoing changes is difficult, it is also very valuable. Proofs encode an infinite number of test cases, and detect violations of invariant requirements that tests cannot. In the Hotel example, we proved that once a new guest enters a room, the previous guest no longer has access, provided the card keys are encoded such they are not identical to or inverse of each other. We then added a new feature, a master key, to the system. Tests passed, including new tests for the master key. But the security property, that the previous guest no longer has access, was violated. The tests did not detect this, but reproving the security theorem failed, revealing the condition where the previous guest's key matched the master key. While we expected counterexample generation to detect this, issues described in section 4.9 prevented this. The failed expression from the proof was enough to require a new assumption be added to the proof and a corresponding requirement added to the card key encoder, to prevent guest keys from being encoded with the master key codes.

## *4.11 Approach to DSLs in General*

As described in the paper presented at the NASA Formal Methods Symposium [12], the FORMED approach applies to domain specific languages in general, not just UML. On a separate project called SITAPS (Specification Improvement through Analysis of Proof Structure), we transformed the Ivory textual DSL, optimized for drone flight control, into ACL2s to prove properties about the functions. While we believe ACL2s with its theorem proving support is a good formal language for verification, mappings to other formal languages such as Coq could also be made.

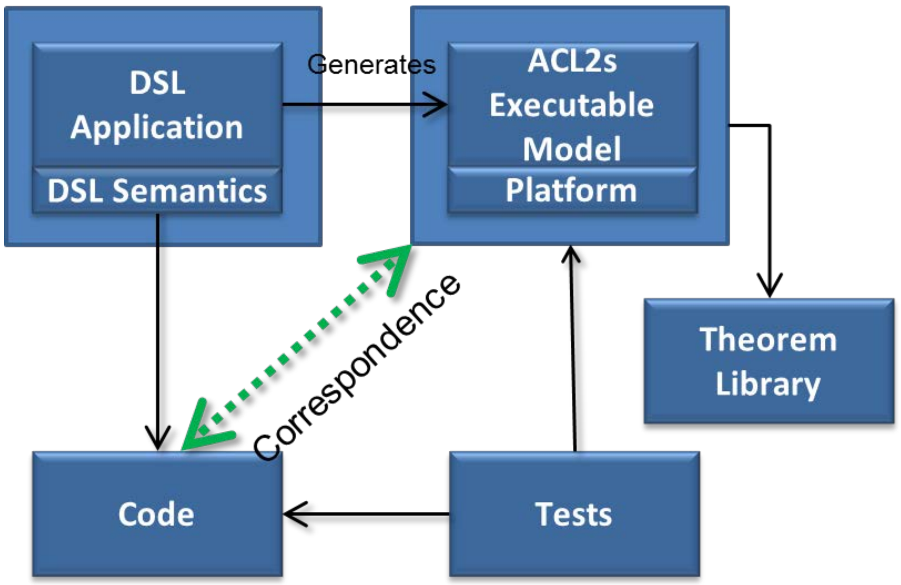

**Figure 10 Domain specific languages (DSLs) drive both implementation and formal verification**

## *4.12 Metrics*

FORMED metrics reports the lines of code and number of theorems both generated and hand-written. This section describes the collection of the metrics.

Lines of Java code were counted using the open source tool JavaNCSS, from http://www.kclee.de/clemens/java/javancss/. Lines of ACL2s were counted by counting the number of open parentheses “(“, since each represents a function call, assignment, or branch condition.

Generated lines of code were kept separate from hand-written lines of code, since regeneration would clobber any hand edits, easing the tracking of hand vs automated code.

No attempt was made to collect effort metrics on theorem code, and, to our knowledge, no effort per line of code metrics have been published for any formal verification language.

Theorem metrics were created by counting the number of *thm, defthm, defthmd* and *defthmt?* statements. Each of the above keywords represents a theorem, but ACL2s handles them each in a different way.

More work remains to evaluate the efficacy of the generated theorems in supporting application level proofs. While some of the theorems supported proving other theorems, some theorems remained unused, and these depended upon the application theorems developed. Where, in the process of proving theorems, we identified required lemmas, we automated the generation of those lemmas where possible. As a result, FORMED generates more theorems than would be required, since lemmas about an association, for example, would be replicated for each association in the model. (Future Work) Research into management of the theorems and lemmas to prove the application level properties would both guide the novice ACL2s user and optimize the verification process.

**Table 5 Metrics of lines of code and theorems show 90% lines generated or reused.**

| | | Hotel Locks | Equip Mgmt | Planner Prototype |
|---|---|---|---|---|
| UML | # Classes | 14 | 33 | 103 |
| | #operations | 30 | 58 | 209 |
| | AL SLOC | 290 | 187 | 1594 |
| Java | Java SLOC | 3197 | 7959 | 27780 |
| Exe ACL2s | Functions | 220 | 335 | 1143 |
| | Defdata | 84 | 98 | 259 |
| | SLOC | 4786 | 6139 | 24118 |
| Auto Semantics Thm ACL2s | Theorems (thm + defunc) | 640 | 960 | 3472 |
| | Theorems SLOC | 8082 | 11578 | 43348 |
| Auto App Thm ACL2s | Theorems (thm + defunc) | 59 | 59 | 228 |
| | Theorems SLOC | 1216 | 1356 | 5243 |
| Hand Thm ACL2s | Theorems (thm + defunc) | 76 | 44 | 0 |
| | Theorems SLOC | 1920 | 704 | 0 |
| Platform | Functions | 62 | | |
| | Function SLOC | 1263 | | |
| | Theorems (thm + defunc) | 198 | | |
| | Theorems SLOC | 1550 | | |
| Summary | Total SLOC | 18817 | 19777 | 72709 |
| | Hand SLOC | 1920 | 704 | 0 |
| | %gen/reuse SLOC | 90 | 96 | 100 |

Table 5 shows lines of code and number of theorems applied to the pedagogical Hotel example, the military equipment management application, and an internal planning prototype developed using UML prior to the FORMED project. The platform code and theorems, consisting mainly of the MddContext, are reused across all examples.

The generated ACL2s code is meant to support theorem proving, and not be an efficient deployable implementation. Extra lines of code were generated (and duplicated) to facilitate theorem proving. Still, the size of the ACL2s code is close to the Java code, although the Java code in the equipment and planner examples includes a large number of XML related code that the ACL2s does not.

While there is no upper bound on the number of hand-written theorems or lines of code, as we identified common patterns and lemmas in our proof effort, we folded those patterns back into the ACL2s code generation. While we expect application theorem proving to be manual while the technique matures, we still expect 90% of the code and theorems to be automatable.

## *4.13 Component Composition with Encapsulate*

Packages in the FORMED UML profile are opaque, hiding the details of their implementation behind a façade interface. Using this strict API, development of a package depends only on API calls to other packages.  FORMED uses an assume-guarantee approach across package APIs to reason about package behavior independently, while also guaranteeing behavior about their composition.

FORMED maps domain façade operations to ACL2s functions, but for operations that are not implemented with the FORMED profile, FORMED generates an ACL2s stub.  Developers can then use ACL2s *encapsulate* to define the properties the function.  Encapsulate defines the function signature and theorems about the function, but requires a simple, hidden implementation of the function in order to prove the properties.  The function can then be called by others, and the properties defined in the encapsulate statement used to support other properties.

The ACL2s code below shows the encapsulate statement wrapping SW_GetCommandLineArg(index), a function to access the parameters passed on the command line to the executable.  The actual implementation of the function differs for ACL2s, Java, and C++, and so is hand-written for each of those target languages.  The interface remains the same, returning the string at the *index* or an empty string if the index does not exist.

```
(encapsulate
 ((SW_GetCommandLineArg (i MddContext)               ; signature
              t
              :guard (and (MddContext-typep MddContext) (natp i))))
 (local (defun SW_GetCommandLineArg (i  MddContext)         ; local example implementation
        (declare (ignore i)
          (xargs :guard (and (MddContext-typep MddContext) (natp i))))
        (SW_GetCommandLineArg_Output "" MddContext)))
 (defthm returnsEmptyStringEventually                        ; property – returns empty string
   (let ((n (length (mget :cmdLineArgs MddContext))))
    (implies (and (natp n)
             (natp i)
             (MddContext-typep MddContext)
             (> i n))
         (equal (length (mget :_retval (SW_GetCommandLineArg i MddContext))) 0))))
 (defthm contract                                             ; property – interface contract
   (implies (and (natp i)
            (MddContext-typep MddContext))
        (SW_GetCommandLineArg_Outputp (SW_GetCommandLineArg i MddContext))))
 (defthm readonly                                   ; property – does not alter the heap or stack
   (implies (and (natp i)
            (MddContext-typep MddContext))
        (equal (mget :MddContext (SW_GetCommandLineArg i MddContext)) MddContext )))
)
```

**Figure 11 ACL2s encapsulate statement defines properties to access command line arguments from the FORMED model**

The signature includes a guard that defines the input parameter types, and these must align with any implementation of the function.  ACL2s requires an example implementation of the function to be able reason about any properties claimed about it, but the function is declared *local* so it is not visible outside of the encapsulate statement.

The theorems in the defthm statements claim that the function returns an empty string when the index is greater than the length of the args list in the MddContext, that it returns a string and the MddContext, and that the MddContext returns unaltered by the function. The local implementation is used to verify that there is at least one function that meets these properties.

These properties are then used to prove termination of the following PAL statements. Since i increases, and we have a property that the function will eventually return an empty string, the while loop will terminate.

```
Integer i = 0;
String xmlFile = "";
WHILE  (xmlFile != "")
{
        xmlFile = SoftwareMechanisms:GetCommandLineArg(i);
        i = i + 1;
}
```

**Figure 12 FORMED code to access command line arguments relies on encapsulated properties to prove termination**

While we can reason about this function, at this point, the function cannot be executed, and results in a run-time error when encountered running a test case. ACL2s *defattach* introduces a function as an implementation of the encapsulated function. Defattach applies the encapsulated function’s theorems and proves that they hold in the implementing function. Defattach also connects the encapsulated function with the implementation during execution, so the implementation function will be invoked, rather than generating a run-time error.

```
(defun SW_GetCommandLineArgImpl (i MddContext)
  (declare (xargs :guard (and (MddContext-typep MddContext) (natp i))))
; implementation omitted
)
(defattach (SW_GetCommandLineArg SW_GetCommandLineArgImpl))
```

**Figure 13 Defattach proves an encapsulated function’s implementation conforms to all its properties**

Encapsulated functions allow independent development of packages and the representation of properties of APIs whose implementations are not modeled with FORMED.

Currently, encapsulate statements, their example implementations, and their theorems must be defined in ACL2s. Future work would allow properties to be expressed in OCL and translated to ACL2s theorems, and example function implementations in action language also translated to ACL2.

## *4.14 State Machine Semantics*

As part of formalizing the executable semantics of the FORMED UML profile, we implemented in ACL2s a set of MddContext platform functions (Table 4) that encapsulated the operation of the state machines and developed some supporting theorems.

FORMED maps the UML state models into ACL2s, supported by the MddContext API. The semantics of the state model behavior include:

- State actions take zero time to execute
- Only one state action is active at a time
- Pending events that have been sent but not yet delivered are stored in a "queue"
- Pending events are handled sequentially, with the following constraints
    - Self-directed events are delivered before events from other sources
    - Events from a source to a destination are in order
    - Otherwise, any pending event may be handled next
- Untriggerred transitions are considered self-directed

State machines, when receiving events, execute the current state's exit action, followed by the transition action, followed by the new state's entry action.

Delayed events serve to handle time dependent processing. Events in action language are sent with a delay, to be handled not before that delay expires. In the ACL2s implementation, the current (logical) time is taken from the MddContext and added to the delay, so the event is placed in the event queue with the expire time.

When choosing the next event to handle, the semantics may choose any of the non-delayed events according to the constraints above, or the next delayed event (with the smallest expire time). If a delayed event is chosen, the current time (within the MddContext) is updated to the expire time of the event, ensuring the correct ordering of delayed events.

The OMG is currently updating its UML specifications to add firmer semantics for state machines. We are working with this group to bring the formal representation of state machines using ACL2s.

## *4.15 Theorem Patterns*

As we applied FORMED to the example problems, we integrated the theorems and lemmas we developed back into the transformation rules. The theorem patterns then generated the theorems everywhere they were applicable to the UML model. We used stereotypes and PathMATE properties as additional information to guide the application of these patterns.

Indeed, as we worked to prove the input-output contracts of the generated functions, we noticed lemmas about associations that were required to prove the contracts. For example, functions that supported the linking and unlinking of objects over associations do not change the stack. We formalized the pattern of these lemmas and added them to the translation rules so the lemmas would be generated for each association. ACL2s then applied the new lemmas supporting the contract proofs.

### 4.15.1 Induction

FORMED generates the infrastructure for inductive proofs. A step function takes parameters instructing what modeled operation to invoke along with parameters. The inputCheck functions described above make sure that parameters that do not conform to the correct data types still return the required (and unchanged) MddContext for the inductive proof. A simplified version of the step function is shown below.

```
(defun induction_step (op params MddContext)
  (case op
        (‘operation1 (operation1 params))
        (‘operation2 (operation2 params)) … ))
```

**Figure 14 FORMED generates the infrastructure for induction proofs**

A run function recursively calls the step function over a list of operations to be executed, with parameters. As free variables within a theorem, the operations and parameters represent all possible invocations of the application's API.

FORMED uses this induction infrastructure for many theorems automatically generated from the model. Some examples of inductive proofs derived from the UML model include association multiplicity, where an object must be related to one or more objects over an association, and unique attribute values across a class extent (all guests must have unique names).

For uniqueness, and other invariant theorems derived from the models, we used static analysis to simplify the proof obligation. Static analysis of the UML model narrows the number of function over which we have to induct. Functions that do not modify the association or attribute that is the basis of the proof can be filtered out of the proof. We do this by generating a proof-specific defdata structure enumerating the list of function to induct, and use it to constrain the operations passed to the induction step function.

### 4.15.2 Uniqueness

Uniqueness, a common property of many UML models, constrains the values of attributes across sets of objects to be unique. For example, no two people can have the same social security number. Uniqueness is a property that we can apply to a model, and using templates, we can generate the theorem for a uniqueness invariant.

The *Identifier* stereotype applies to attributes within a set, either a class extent or across an association that must be unique within that set. Figure 15 shows a Hotel associated with one or more Rooms, each of which must have a unique number. Another property, not shown here, captures the extent or association for the uniqueness property.

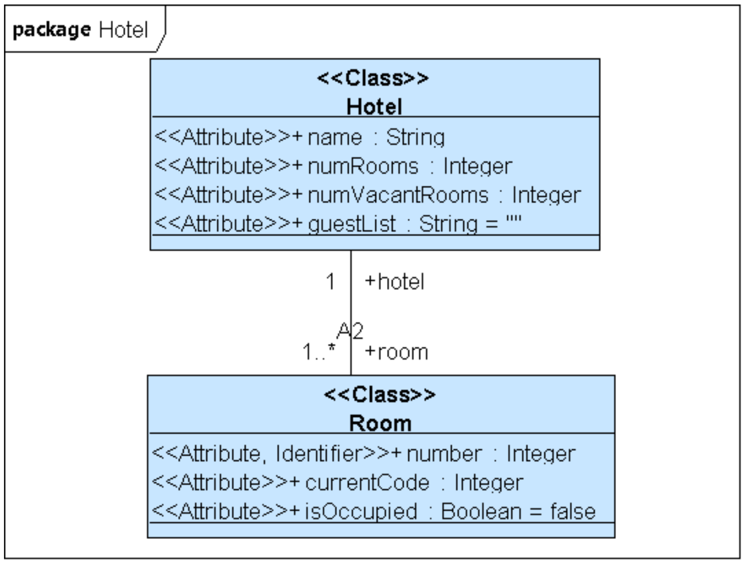
package Hotel
<<Class>>
Hotel
<<Attribute>>+ name : String
<<Attribute>>+ numRooms : Integer
<<Attribute>>+ numVacantRooms : Integer
<<Attribute>>+ guestList : String = ""
1
+hotel
A2
1..*
+room
<<Class>>
Room
<<Attribute, Identifier>>+ number : Integer
<<Attribute>>+ currentCode : Integer
<<Attribute>>+ isOccupied : Boolean = false

**Figure 15 Room number is tagged as an *Identifier*, indicating its value must be unique**

Uniqueness is expressed as an invariant and uses induction to prove it holds for all execution paths through the system. The theorem states that if the system starts in a state where uniqueness holds, then any possible set of traces through the system will result in a state where the invariant also holds.

### 4.15.3 FSM fairness

Proof efforts focused on the properties of communicating finite state machines. Some of these properties require reasoning about the traces, or sequences of events, between the state machines. ACL2s has limited support for quantification reasoning with traces, such as identifying the traces that can reach an unsafe state using an exists quantifier. Rather, with ACL2s we must reason about all the traces.

Theorems about the traces of events handled by the set of communicating finite state machines must include properties about the traces themselves. In the case of the FORMED UML models, state machines are reacting to events generated elsewhere in the application, and generate other events to other objects in the model, while also dealing with external events from outside the model, such as timeouts. Fairness is a property that these traces must possess; otherwise one or two objects may dominate the processing and prevent other objects from executing. This notion is similar to thread scheduling in operating systems. A fairness function constrains both the internal event orderings and allowable external event injection to support reasoning about the state machine interactions. This fairness function becomes an assumption about the deployment environment that must be examined in addition to correspondence testing.

### 4.15.4 Read-only

UML operations that access data or perform computations and do not modify the state of the system (the heap) are *read-only*. The property makes a useful lemma when trying to prove other properties, perhaps about a function that calls a read-only function. The UML operation is marked using the ReadOnly property (see the User's Guide), and the theorem is generated from the template.  Read-only implies that the function does not change the stack, heap, or event queue of the system.

### 4.15.5 Operation Contracts

Operation contracts define the input and output contracts of a function.  The contracts mostly conform to the data types passed in and out, but can include other pre- and post-conditions.

Contracts can be expressed using the ACL2s *defunc* construct, similar to *defun* that defines a function.  Defunc captures the input and output as separate expressions, and evaluates the contracts against the body of the function.  The function is admitted if all paths through the function satisfy the contract.  Additionally, defunc will attempt to prove that the function satisfies the input contracts of any functions it calls.

The MddContext platform functions are all defined using defunc, and include input contracts that prevent them from operating on nil object references, the FORMED equivalent to NULL pointers in Java or C.  Unsatisfied input contracts at the platform level prevent theorems at the application level from being successfully proven.  Thus, once an application function contract is proven, there are no NULL pointer exceptions possible in the implementation.

FORMED uses *defun* instead of the *defunc* for defining functions.  Defunc sometimes requires lemmas to prove the contract holds, but the code generates the executable functions into a file separate from the theorems.  Therefore, we implemented the function input-output contracts as separate theorems in the <model>-operations-contract.lisp file.  The theorems still evaluate and guarantee no NULL pointer exceptions.

## *4.16 Action Language – ALF vs PAL*

FORMED builds on the PathMATE and Topcased tools for UML modeling and transformation. PathMATE uses a dialect of action language called PAL, platform-independent action language. The OMGs standard syntax for model level action language, ALF, is not yet fully supported by tools.  Both ALF and PAL conform to the action language semantics first introduced into UML 1.5 and carried into UML 2.x.

Service handles are an extension PathMATE made to the action language.  Service handles essentially allow pointers to UML operations to be passed around and invoked later, usually as callbacks in response to some event.  Since ACL2s does not support higher order functions, we implemented service handles as a function name going through a dispatch function to make the invocation.  However, we restricted the operations invoked this way to not invoke any service handles themselves, to avoid mutual recursion conflicts.

## *4.17 Extensions to ACL2 Sedan*

This section describes the extensions that were made to ACL2s as part of the FORMED effort.

### 4.17.1 Bug Fix

While attempting to prove theorems about the example models, we discovered and fixed a soundness defect within ACL2s that resulted in incorrect theorems being proven and accepted.

This defect was found during reasoning about one of the function that supports the UML semantics in FORMED. The ACL2s termination analysis recorded wrong "measured variable" for a recursive function definition introduced using defunc. This resulted in ACL2s inferring an unsound induction scheme based on the function definition.

### 4.17.2 Defthmt?

FORMED is expected to generate a number of theorems that initially may not pass, or, as development progresses, have theorems that once passed, fail due to changes to the UML model. For the convenience of the developer, we created *defthmt?(* The "?" is part of the name of the construct and not an operator), a new theorem definition statement that overrides the default stop-on-fail behavior of ACL2s, and continues processing to identify as many issues as possible.

Defthmt? specifies a timeout, in seconds, after which the theorem is deemed to have failed. Some theorems cause ACL2s to "hang" and not complete the processing, while others just take a long time. Defthmt? will treat both of these the same if they exceed the maximum time. Failing theorems are also flagged, but in either case, ACL2s processing continues. The statement *(table defthmt-failure-table)* reports on the failing theorems and should be placed at the end of any files using defthmt?.

```
(defthmt? Hotel_hasCurrentKey-is-read-only
  (let* ((fnOut (Hotel_hasCurrentKey name room before))
         (after (mget :MddContext fnOut))
        )
        (implies (and ;(MddContext-typep after)
                      (MddContext-typep before)
                      (equal (mget :exception before) nil)
                      (equal (mget :exception after) nil)
                      (Hotel_hasCurrentKey_Outputp fnOut)
                      (equal (Hotel_hasCurrentKey_inputCheck name room before) t))
                 (equalContexts after before)))
  :time 60  )
…
(table defthmt-failure-table)
```

**Figure 16 *Defthmt?* Continues processing ACL2s statements after theorem failures or timeouts**

For theorems that pass, *defthmt?* doubles the time to process the theorem, first to check that it will pass, and second to admit the theorem.

### 4.17.3 Counterexamples

Since the default ACL2s counterexample generation did not work for FORMED's MddContext data structure or support the bidirectional association relations between objects on the heap, we designed our own counterexample generation algorithm, described insection 4.9.

We also needed a new build of ACL2 and ACL2s to allow us to replace the default cgen algorithm with our own, using *(defdata-attach :enumerator)*.

### 4.17.4 Contradictions in Hypothesis

Early attempts at ACL2s theorems, examples of novice usage, included theorems that contained contradictions in their hypothesis. These contradictions come about through the complex hypotheses and the reasoning about nested functions, where one clause assumes n to be a natural number and another assumes n is nil. Since false implies true, these contradiction allow any conclusion to be proven, including *nil* (or false) which should never be able to be proven.

So a lesson learned, that when a proof seems too good to be true (pun intended), or passes to easily, replace it with nil. If it still passes, there is a contradiction in the hypothesis. We considered implementing an ACL2s macro to identify contradictory hypotheses, but deferred to the simpler "try to prove nil" detection strategy.

## *4.18 Obstacles to Software Theorem Proving*

### 4.18.1 Null Pointers

One of the most common problems to proving even simple contracts is the assumption about the existence of objects when traversing an association or searching a class extent (the set of instances of a class). For example, the code below assumes that an instance of the class EwAsset exists in the extent with the attribute *self* set to true.

```
Ref<EwAsset> self = FIND CLASS EwAsset WHERE (EwAsset.self == TRUE);
GENERATE EwAsset:GetId() TO (self);
```

If no such instance exists, an error occurs, as sending an event to an instance that does not exist violates the platform semantics of the event queue.

These assumptions may depend upon the initialization of the component to create the instance, but this leads to undocumented coupling. A change to the initialization may introduce a defect in this code.

Traversing an unconditional association (such as a Room must be contained within a Hotel), where at least one instance must be linked on the other side, introduces similar issues. Here, we have the unconditional property of the association, which must be respected by all operations. The association has an invariant proof across all action language constructs that operate (LINK, UNLINK, DELETE) on the association. Each operation that operates on the association is part of the proof of the association's unconditional invariant.

Even with the proven invariant property that a traverse of the association will always yield a non-null reference, it is difficult to pull that property into another proof. An assumption, when added to the theorem, may state the unconditional property, but that statement is disconnected from invariant proof, and could lead to incorrect foundation for other properties.

To handle both issues, we encourage the use of safe programing of model actions to include checks for null references when doing extent lookups of association traversals. While the translation rules could automatically introduce the checks, the rules could not always define what the remedial action should be.

### 4.18.2 Heap Size

The size of the heap was unbounded for the purposes of this research. Further work would be needed to reason about the maximum number of bytes and if memory could be exhausted during execution. This analysis would need to include a sizing of each primitive data type and UML object. Memory fragmentation would require additional detailed modeling about the memory manager. (Future Work)

### 4.18.3 Platform Constraints – Data Types

The size of integer and rational numbers in FORMED, under ACL2s, is infinite. Further work can be done to analyze the platform constraints of 32 bit integers or 64 bit doubles on the application behavior, including overflow and underflow conditions. (Future Work)

## *4.19 Correspondence Testing*

The code generator that produces the Java and ACL2s code does not claim to produce artifacts that are correct by construction. No refinement proofs prove either implementation is a correct implementation of the model (though that may be an area for future work) though both the specification and executable code are generated from the same model.

FORMED takes a more pragmatic approach, using testing to confirm that both the ACL2s and Java implementation produce the same results, i.e., they correspond, and thus proof results apply to the source code (which cannot be proven for all cases) and testing results apply to the specification (which is part of the correspondence observation).

State machines introduce non-determinism into the correspondence between Java and ACL2s implementations. Each implementation may choose among multiple pending events, and may choose different events to process, leading to different results. Coordination between implementations is required to ensure each operates on the same event. The Java implementation may send notification of its choice to the ACL2s so it may choose the corresponding event. We did not use this coordination, but it is described in other work [13].

Tests are implemented as part of the model and translated into both implementations. Tests are marked using the VerificationArtifact property to allow them to be removed for deployment. Test case generation is manual for FORMED work, but many model-based test case generation tools are available [14].

Derivation of tests from the proofs is an area for future work. Test cases that cover each step in the proof might be able to prove that the equivalent behavior exists in the implementation as the specification. (Future work)

## *4.20 OCL*

After the initial mapping of UML to ACL2s, we focused on developing and proving theorems at the UML semantics and application levels. Since we were able to express the application level theorems directly in ACL2s, we postponed implementing the Object Constraint Language (OCL) [15] to ACL2s transformation, as it was lower risk than defining and proving theorems. OCL, a specification closely related to UML, expresses declarative constraints on UML models, such as preconditions, postconditions, invariants and association constraints.

As UML model elements are mapped to ACL2s, OCL constructs map to ACL2s functions and theorems. OCLS collections and extents map easily down to the extents collected from the heap, the lists that support associations, and other list constructs. (Future Work)

# 5 Conclusions and Recommendations

FORMED integrates formal verification into software design and development by precisely defining semantics for a restricted subset of the Unified Modeling Language and transforming application models into both an ACL2s formal specification for analysis and Java code for deployment. Correspondence testing verifies consistent translation and executable behavior between the formal and deployed implementations.

FORMED creates an IDE integrating the UML and ACL2s tools under the Eclipse framework. The integration provides the connection between the UML model and the generated ACL2s code to assist the application developer in understanding the specification code and in learning the ACL2s environment. FORMED envisions the application developer worked together with a verification expert to perform the proofs that are not automated. ACL2s takes time for novices to learn, and working within the framework and an expert reduces the learning curve.

We were able to automate generation of a number of theorems and automated a subset of their proofs. Operation input-output contracts ensure no null pointer references. Termination proofs verify that no infinite loops or live-lock conditions exist. Theorems verify that UML operations respect invariants such as unconditional relationships and unique attribute values across sets of objects.

To reduce or eliminate the visibility of the formal verification tools more research is required to identify proof patterns and the useful lemmas on top of the UML semantics and MddContext platform. Not all lemmas support application properties, and too many lemmas make the knowledge space ACL2s must deal with too large. Generation of lemmas should be balanced with the management of the active theorems to guide the ACL2s heuristics in proving theorems. This management requires determining the proof strategy from the type of theorem and from the UML models.

## *5.1 Application to High Assurance Software Development*

The application of formal verification to high assurance software development needs to be explored further. Current high assurance processes, such as DO178c allows for formal methods, but as an add-on. All other DO178c procedures are unchanged.

To incorporate a formal verification tool tightly into the high assurance toolchain, the standards require that tool to be developed with the same requirements and procedures as the high assurance software itself. Unfortunately, this is not feasible with formal verification tools that typically evolve out of university research. This makes formal methods looks as just extra work with no value added, but does not address the beneficial impact of formal verification on the other DO178c steps.

Many current high assurance development methodologies suffer from one of the following flaws:

- The proofs are done at the design level but not carried through to the implementation, relying on manual processes to implement the specification's properties.
- The implementation is derived from formal design specification, but is an inefficient implementation and may not integrate well with other components.
- An efficient implementation is derived but the overall scope of the specification methodology is limited and not sufficient for general application development.

FORMED provides an approach where design level proofs apply to generated code (via correspondence), but overall implementation is performed in a software engineering environment which supports production of code that fits (i.e., it is efficient and integrates into) the overall software engineering effort.

# 6 Bibliography

1. Jackson, Daniel, Software Abstractions, MIT Press, 2011.
2. SPECWARE - Producing Software Correct by Construction. James McDonald and John Anton. Kestrel Institute Technical Report KES.U.01.3., March 2001.
3. http://www.omg.org/spec/ALF/
4. Balcer, Marc and Mellor, Steve, Executable UML: A Foundation for Model-Driven Architectures, Addison-Wesley Longman Publishing Co., Inc. Boston, MA. 2002.
5. http://www.pathmate.com
6. Li, Junsong, et. al., Slimming Languages by Reducing Sugar: A Case for Semantics-Altering Transformations, to appear in Onward 2015.
7. Eastlund, Carl, Dale Vaillancourt, and Matthias Felleisen. "ACL2 for freshmen: First experiences." Proc. 7th Intern. ACL2 Symposium. 2007.
8. http://www.omg.org/spec/UML/
9. Moore, J. Strother. "Stateman: Using Metafunctions to Manage Large Terms Representing Machine States." ACL2 Workshop, 2015.
10. Liu, H and Moore, J. Java Program Verification via a JVM Deep Embedding in ACL2, Theorem Proving in Higher Order Logics (TPHOLS '04), 2004.
11. Wiegley, John and Reubenstein, Howard, "Formalizing a Register Allocator in Coq", submitted to Conference on Certified Programs and Proofs, 2016.
12. Eakman, Greg, et. al., "Practical Formal Verification of Domain Specific Language Applications", Proceedings, 7th Annual NASA Formal Methods Symposium, Pasadena, CA, 2015.
13. Eakman, Gregory, T. A Systems Approach to Testing Object-Oriented Models, Boston University, 2002.
14. http://www.agileconnection.com/sites/default/files/article/file/2012/XDD6047filelistfilename1_0.pdf
15. http://www.omg.org/spec/OCL/

## LIST OF ACRONYMS, ABBREVIATIONS, AND SYMBOLS

| | |
|---|---|
| ACL2 | A Computational Logic for Applicative Common Lisp |
| ACL2s | ACL2 Sedan |
| ALF | Action Language for Foundational UML |
| API | Application Program Interface |
| AST | Abstract Syntax Tree |
| DSL | Domain Specific Language |
| FORMED | Formal Methods Engineering Desktop |
| IDE | Integrated Development Environment |
| OCL | Object Constraint Language |
| OMG | Object Management Group |
| OO | Object Oriented |
| PAL | Platform-independent Action Language |
| PathMATE | Path Model Automation & Transformation Environment |
| SITAPS | Specification Improvement through Analysis of Proof Structure |
| UML | Unified Modeling Language |
| UUID | Universally Unique Identifier |
| XML | EXtensible Markup Language |